



Multicore Aware Data Transmission Middleware (MDTM)

Interim Progress Report
09/01/2013 --- 03/31/2014

Fermi National Accelerator Laboratory
Batavia, Illinois, USA

Brookhaven National Laboratory
2 Center St, Upton, NY 11973

Table of Contents

1. Summary	3
2. Project Overview	4
2.1. The problem	4
2.2. Our solution: a Multicore-Aware Data Transfer Middleware (MDTM).....	4
2.3. Major Milestones.....	5
3. Accomplishments	6
3.1. Application	6
3.1.1. Major Activity and Progress.....	6
3.1.2. Significant Results	6
3.2. Middleware	7
3.2.1. Major Activities and Progress	7
3.2.2. Significant Results	8
3.3. Integration.....	11
3.4. Collaborating Environment	13
3.4.1. Major Activities and Progress	13
3.4.2. Significant Results	14
4. Next Steps.....	14
Appendix A: MDTM Application Example Configuration File	15
Appendix B: MDTM Middleware Example Outputs	16
Appendix C: MDTM Collaborating Environment	18
Appendix D: MDTM Library API Functions.....	19

1. Summary

To date the project has achieved all of its targets for this reporting period. These include,

- Completion of thread/flow management module*
- Completion of phase-I preprocessing module*
- Completion of MDTM middleware APIs*
- Completion of multicore system profiling module*
- Completion of topology-based resource scheduler*
- Completion of interrupt affinity for network IO*
- Integration of MDTM middleware library and data transfer application*
- Establishment of collaborating environment including Redmine, GIT, Share point and pubic website*

The weekly work meeting was well attended by members from across the DOE national laboratories, including the BNL and FNAL. The project is currently moving toward delivering the first release in late July 2014.

2. Project Overview

2.1. The problem

Multicore and manycore have become the norm for high-performance computing. These new architectures provide advanced features that can be exploited to design and implement a new generation of high-performance data movement tools. To date, numerous efforts have been made to exploit multicore parallelism to speed up data transfer performance. However, existing data movement tools are still bound by major inefficiencies when running on multicore systems for the following reasons:

- Existing data transfer tools are unable to fully and efficiently exploit multicore hardware under the default OS support, especially on NUMA systems.
- The disconnection between software and multicore hardware renders network I/O processing on multicore systems inefficient.
- On NUMA systems, the performance gap between disk and networking devices cannot be effectively narrowed or hidden under the default OS support.
- Data transfer tools receive only best-effort handling for their process threads. There is no differentiation in service based on transfer characteristics, thread locality needs, or prioritization requirements.

These inefficiencies are fundamental and common problems that data movement tools will inevitably encounter when running on multicore systems. These inefficiencies will ultimately result in performance bottlenecks on the end systems. Such end system performance bottlenecks also impede the effective use of advanced networks. The DOE ANI (advanced network initiative) deployed 100-gigabit WAN testbed in support of the next-generation distributed extreme-scale data movement. Resolving performance issues within computer hosts is becoming the critical element within the end-to-end paradigm of the distributed extreme-scale data movement.

2.2. Our solution: a Multicore-Aware Data Transfer Middleware (MDTM)

MDTM aims to accelerate data movement toolkits at multicore systems. Essentially, MDTM consists of two research components (Figure 1):

- MDTM data transfer applications/tools research and development
- MDTM middleware research and development

For the MDTM project, we plan to achieve the following research goals:

- To develop and optimize ultra high-speed data transfer applications/tools on modern multi-core systems.
- To investigate, design, and implement generic middleware mechanisms to enable extreme-scale data movement tools to exploit the multicore hardware fully and efficiently, especially on NUMA system.

- To deploy, test, and comprehensively evaluate the developed middleware/applications, on advanced multicore hosts, and over 100Gbps+ testbed networks.

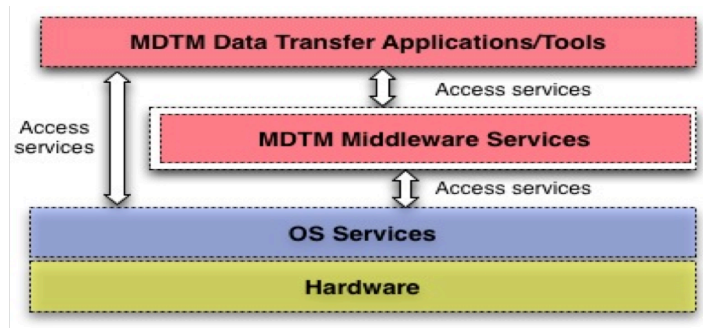
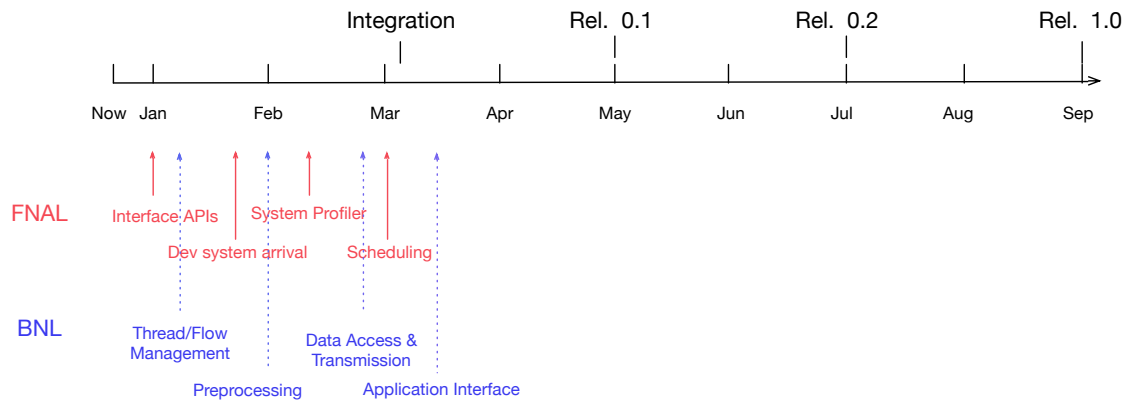


Figure 1. MDTM Software Architecture

MDTM will be deployed in DOE data transfer nodes (<http://fasterdata.es.net/science-dmz/DTN/>).

2.3. Major Milestones

The major milestones of the MDTM project are set as follows.



3. Accomplishments

3.1. Application

3.1.1. Major Activity and Progress

Thread/flow management module (completed). Parallel threads can be created and executed for both storage and network devices. Among these threads, communication and synchronization are implemented to ensure the orderly execution. Buffer memory can be created to ensure the data exchanged between threads (e.g., storage reader and network sender, using a producer-consumer model). In addition, we can automatically creating multiple threads based on capacity of NICs and storage devices, e.g., for more processing threads needed for high-capacity devices.

Preprocessing module (phase I completed). In our design, we group the requested files by devices (HDDs, SSDs, hardware/software RAID, SAN, etc), so that the data transfer application can maximize the locality of access. Currently, the basic function of grouping is implemented. After that, we can now dynamically creating threads for groups (one or multiple threads for each group, also based on the type and capacity of devices).

An additional work we have done is to create a system configuration step (and configuration file). The configuration file lists information on the device type, mappings (logical to physical devices), and capacity. Our application can read, parse, and analyze the devices, and then using this information to dynamically create threads.

3.1.2. Significant Results

Parallel storage threads. In the BBCP software framework, multiple storage threads are created to execute parallel I/O on different storage devices.

Parallel network threads. In the BBCP software framework, multiple network threads are created to execute parallel transmission.

Grouping of files: The application can create multiple groups of requests, and schedule different threads for them.

MDTM Application protocols for client/server: We specify and verify the protocols between the MDTM application client and server. The protocols determine the behavior of the participants in various stages, including the initialization, preprocessing, and thread/flow management. The interaction in the preprocessing stage is shown below.

System Configurations: The application can obtain device mapping information from system configuration file provided by user. An example configuration file is in Appendix A.

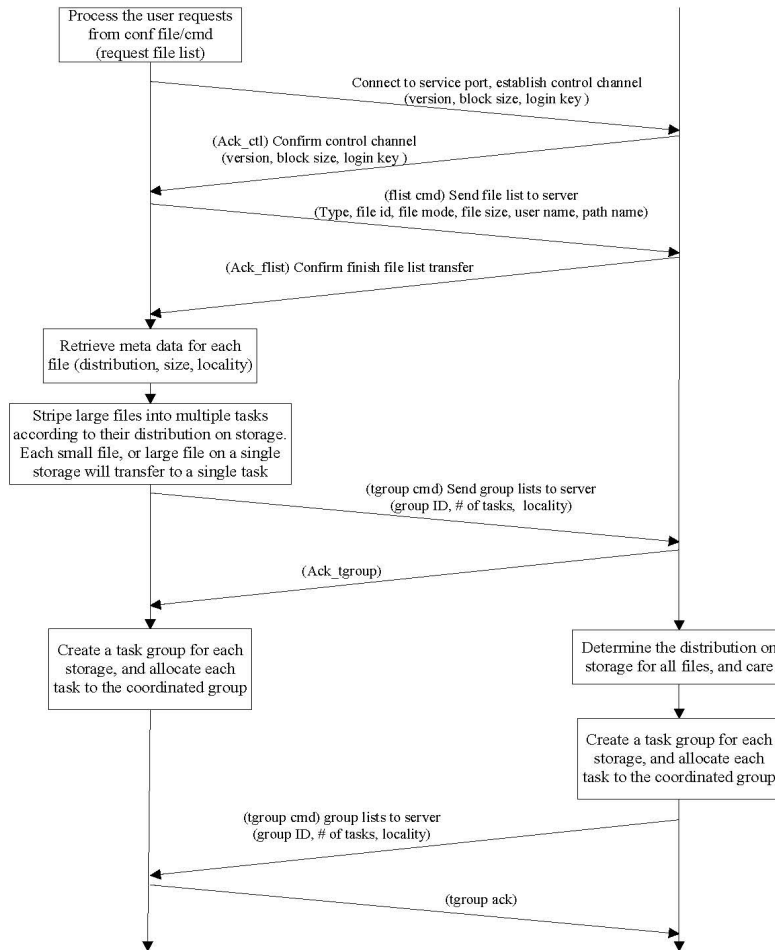


Figure 2. MTDM application protocol in the preprocessing stage

3.2. Middleware

3.2.1. Major Activities and Progress

MDTM middleware APIs: This part was successfully completed and has already been validated by the MDTM application during the integration phase. Those APIs was carefully designed to provide easy to use interfaces for any applications that need to leverage the multicore and NUMA architectures to achieve high performance in terms of throughput and latency. They contain rich functions, which fall into four categories: system information, thread management, IO interrupt management and utilities. Those APIs hide all multicore system caveats from applications, which can focus on their own business logic.

System profiling module: This part was successfully completed and already used by the MDTM application to create system table. The profiling module intensively interacts with the Linux kernel services like *procfs*, *sysfs*, device drivers, virtual file system and etc. With that profound Linux kernel knowledge, the profiling module retrieves valuable system information including components in place (CPUs, NUMA nodes, memories, PCI devices, NIC, disk), system topology (affinity and distances between components), working status (component identity and activity), metrics (IO bandwidth and disk capacity) and etc. Another key implementation in the profiling module is the so-called “MDTM tree” which is used to store and organize pieces of scattered system information. With the well-structured “MDTM tree”, any request for system information from application can be served in a very efficient and complete way. In addition, the profiling module is also scalable by nature since the tree can be easily extended to cover more information interested in the future.

Topology based resource scheduler: This part was successfully completed and deployed in the MDTM application. The research work carried here shows the locality in terms of CPUs, IOs and threads obviously improves applications’ performance over multicore system since it bridges the gap between conventional Linux scheduler and modern multicore architectures. Upon receiving service requests from MDTM application threads like the Readers, Writers, Receivers and Senders, the MDTM scheduler should look for and deploys those threads to CPUs that are close in distance to storage disks and network interfaces in order to maximize the data throughput. In addition, working loads and traffic conditions between system components especially the NUMA nodes also affect the overall performance. The MDTM scheduler takes initiatives to abstract the complicated system interconnection as a graph and associates each connection with a cost value that reflects scheduling factors like distance, traffic condition and etc. By applying searching algorithms like Dijkstra’s algorithm, the lowest cost path from specific CPUs to the targeted NIC/Disks is found and therefore be used as the candidates to run Reader/Writer/Sender/Receiver threads.

Interrupt affinity: This part was successfully completed for network adapters. Multiple queue (MQ) and RSS are widely used in today’s high-speed network adapters. The MDTM middleware leverages the RSS and flow director by setting the affinity of specific data flow to its associated application thread. Therefore it improves the throughput and maximizes the parallel processing of the network traffic.

3.2.2. Significant Results

Some major results are reported here. More results can be found in the Appendix.

System Topology Tree: MDTM profiling module creates an internal MDTM tree that contains the topology and detailed information of the multicore system. Using MDTM API function generates *Figure 3*, which shows the topology tree of our testing system with eight cores, two NUMA nodes, two network adapters and two disks.

CPU and Device Affinity: Figure 4 shows the affinity of the testing system generated and output by the MDTM profiling module.

MDTM Scheduling: Figure 5 presents the output of MDTM scheduler from traversing the testing system. The cost values to specific IO devices are calculated and potential CPUs are found to running the thread dealing with data transferring between those devices.

Interrupt Affinity: Figure 6 shows the result of interrupt affinity assignment by the MDTM middleware for the network interface adaptor on the testing system. The MDTM middle optimizes the traffic throughput by leveraging the RSS and associate receiving queues to CPUs generating the traffic flows.

```

[liangz@capecod build]$ src/libmdtm/tests/test --topology
=====
System Topology Tree
=====
Machine L#0
  NUMANode L#0 local=67075376KB total=67075376KB
  Socket L#0 CPUModel="Intel(R) Xeon(R) CPU E5-2680 @ 2.70GHz"
  Core L#0
  PU L#0
  Core L#1
  PU L#1
  Core L#2
  PU L#2
  Core L#3
  PU L#3
  Core L#4
  PU L#4
  Core L#5
  PU L#5
  Core L#6
  PU L#6
  Core L#7
  PU L#7
  Bridge Host->PCI L#0 buses=0000:[00-06]
  Bridge PCI->PCI L#1 busid=0000:00:11.0 id=0006:1d3e class=0604(PCI_B) buses=0000:[04-04]
  PCI 0006:1d6b L#0 busid=0000:04:00.0 class=0107(SAS)
  Bridge PCI->PCI L#2 busid=0000:00:1c.0 id=0006:1d10 class=0604(PCI_B) buses=0000:[05-05]
  PCI 1503:1003 L#1 busid=0000:05:00.0 class=0200(Ether)
  Bridge PCI->PCI L#3 busid=0000:00:1e.0 id=0006:244e class=0604(PCI_B) buses=0000:[06-06]
  PCI 102b:0532 L#2 busid=0000:06:03.0 class=0300(VGA)
  PCI 0006:1d02 L#3 busid=0000:00:1f.2 class=0106(SATA)
  Block L#0 sda
  Block L#1 sdb
  NUMANode L#1 local=67108864KB total=67108864KB
  Socket L#1 CPUModel="Intel(R) Xeon(R) CPU E5-2680 @ 2.70GHz"
  Core L#8
  PU L#8
  Core L#9
  PU L#9
  Core L#10
  PU L#10
  Core L#11
  PU L#11
  Core L#12
  PU L#12
  Core L#13
  PU L#13
  Core L#14
  PU L#14
  Core L#15
  PU L#15
  Bridge Host->PCI L#4 buses=0000:[00-04]
  Bridge PCI->PCI L#5 busid=0000:00:01.0 id=0006:3c02 class=0604(PCI_B) buses=0000:[01-02]
  PCI 0006:1521 L#4 busid=0000:01:00.0 class=0200(Ether)
  Network L#2 eth1 Address=00:25:90:99:a0:54
  PCI 0006:1521 L#5 busid=0000:01:00.1 class=0200(Ether)
  Network L#3 eth2 Address=00:25:90:99:a0:55
[liangz@capecod build]$

```

Figure 3. System Topology Tree

```
liangz@capecod build$ src/libmdtm/tests/test --cpuaffinity
=====
Device      CPU set in affinity
=====
eth1:       0 9 10 11 12 13 14 15
eth2:       8 9 10 11 12 13 14 15
sda:        0 1 2 3 4 5 6 7
sdb:        0 1 2 3 4 5 6 7
[liangz@capecod build]$

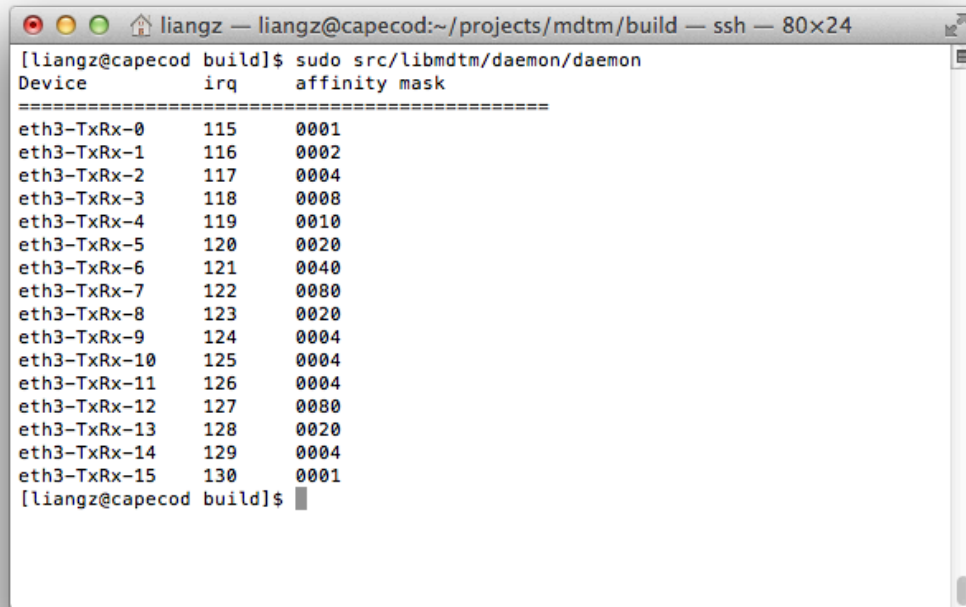
liangz@capecod build$ src/libmdtm/tests/test ---deviceaffinity
=====
Core  Device
=====
0     sda sdb
1     sda sdb
2     sda sdb
3     sda sdb
4     sda sdb
5     sda sdb
6     sda sdb
7     sda sdb
8     eth1 eth2
9     eth1 eth2
10    eth1 eth2
11    eth1 eth2
12    eth1 eth2
13    eth1 eth2
14    eth1 eth2
15    eth1 eth2
[liangz@capecod build]$
```

Figure 4. CPU and Device Affinity

```
liangz@capecod build$ src/libmdtm/tests/test --sched_spt eth1 sda
=====
Shortest Path Tree (SPT) Scheduling
=====
Device CPU/Cost
eth1    0:26 1:26 2:26 3:26 4:26 5:26 6:26 7:26 8:6 9:6 10:6 11:6 12:6 13:6 14:6 15:6
sda     0:5 1:5 2:5 3:5 4:5 5:5 6:5 7:5 8:25 9:25 10:25 11:25 12:25 13:25 14:25 15:25

CPUs  Costs  Path
0     31    (Core 0) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
1     31    (Core 0) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
2     31    (Core 1) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
3     31    (Core 2) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
4     31    (Core 2) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
5     31    (Core 3) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
6     31    (Core 3) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
7     31    (Core 4) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
8     31    (Core 4) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
9     31    (Core 5) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
10    31    (Core 5) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
11    31    (Core 6) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
12    31    (Core 6) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
13    31    (Core 7) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
14    31    (Core 7) ==> (Socket 0) ==> (NUMANode 0) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
15    31    (Core 8) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
16    31    (Core 8) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
17    31    (Core 9) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
18    31    (Core 9) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
19    31    (Core 10) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
20    31    (Core 10) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
21    31    (Core 11) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
22    31    (Core 11) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
23    31    (Core 12) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
24    31    (Core 12) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
25    31    (Core 13) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
26    31    (Core 13) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
27    31    (Core 14) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
28    31    (Core 14) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
29    31    (Core 15) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 4) ==> (Bridge PCI->PCI 5) ==> (PCI 0086:1521 4) ==> (eth1)
30    31    (Core 15) ==> (Socket 1) ==> (NUMANode 1) ==> (Bridge Host->PCI 0) ==> (PCI 0086:1d02 3) ==> (sda)
[liangz@capecod build]$
```

Figure 5. Scheduling Result



```
liangz — liangz@capecod:~/projects/mdtm/build — ssh — 80x24
[liangz@capecod build]$ sudo src/libmdtm/daemon/daemon
Device          irq    affinity mask
=====
eth3-TxRx-0     115    0001
eth3-TxRx-1     116    0002
eth3-TxRx-2     117    0004
eth3-TxRx-3     118    0008
eth3-TxRx-4     119    0010
eth3-TxRx-5     120    0020
eth3-TxRx-6     121    0040
eth3-TxRx-7     122    0080
eth3-TxRx-8     123    0020
eth3-TxRx-9     124    0004
eth3-TxRx-10    125    0004
eth3-TxRx-11    126    0004
eth3-TxRx-12    127    0080
eth3-TxRx-13    128    0020
eth3-TxRx-14    129    0004
eth3-TxRx-15    130    0001
[liangz@capecod build]$
```

Figure 6. Interrupt Affinity

3.3. Integration

We have implemented the MDTM application modules in the BBCP software framework. The adoption of this framework is due to several reasons. First, BBCP have a clean multi-threaded design, and it allows us to dynamically add network and storage threads. Second, BBCP takes an object-oriented C++ implementation, and it has good software modularity. We can then conveniently add our function modules into the framework. Third, we hope to evaluate our MDTM application using standard software tools such as BBCP and GridFTP. Thus it is a good idea to start our implementation with it. In the future, we also plan to integrate the MDTM application with our previous RFTP software toolkit.

After our implementation, we can then have a sample test run of it. Below we show the steps of an example test scenario, just to demonstrate the basic functions of the current software version.

- 1) Control agent get configuration information from command line parameters.
- 2) Control agent forks a source node and a sink node, and these two processes log onto the source and sink sides using ssh respectively.
- 3) Both source and sink nodes call *mdtmApp_Init()* interface to get system topology and save it into a tree structure. This step will call the MDTM middleware interface *mdtm_init()*, which will initialize the *mdtm*

middleware, and `mdtm_create_sys_info(mdtm_node_t* tree)` to get system topology tree.

```

*****
Source side I/O device affinity list
Device(32)          CPU set in affinity
*****
eth7:                0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
ib0:                 0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
mlx4_0:              0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
eth0:                0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
eth1:                0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
sdy:                 0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
eth8:                8 9 10 11 12 13 14 15 40 41 42 43 44 45 46 47
ib1:                 8 9 10 11 12 13 14 15 40 41 42 43 44 45 46 47
mlx4_1:              8 9 10 11 12 13 14 15 40 41 42 43 44 45 46 47
sdax:                16 17 18 19 20 21 22 23 48 49 50 51 52 53 54 55
eth4:                16 17 18 19 20 21 22 23 48 49 50 51 52 53 54 55
eth5:                16 17 18 19 20 21 22 23 48 49 50 51 52 53 54 55
mlx4_2:              16 17 18 19 20 21 22 23 48 49 50 51 52 53 54 55
eth6:                24 25 26 27 28 29 30 31 56 57 58 59 60 61 62 63
ib2:                 24 25 26 27 28 29 30 31 56 57 58 59 60 61 62 63
mlx4_3:              24 25 26 27 28 29 30 31 56 57 58 59 60 61 62 63
eth7:                0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
ib0:                 0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
mlx4_0:              0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
eth0:                0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
eth1:                0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
sdy:                 0 1 2 3 4 5 6 7 32 33 34 35 36 37 38 39
eth8:                8 9 10 11 12 13 14 15 40 41 42 43 44 45 46 47
ib1:                 8 9 10 11 12 13 14 15 40 41 42 43 44 45 46 47
mlx4_1:              8 9 10 11 12 13 14 15 40 41 42 43 44 45 46 47
sdax:                16 17 18 19 20 21 22 23 48 49 50 51 52 53 54 55
eth4:                16 17 18 19 20 21 22 23 48 49 50 51 52 53 54 55
eth5:                16 17 18 19 20 21 22 23 48 49 50 51 52 53 54 55
mlx4_2:              16 17 18 19 20 21 22 23 48 49 50 51 52 53 54 55
eth6:                24 25 26 27 28 29 30 31 56 57 58 59 60 61 62 63
ib2:                 24 25 26 27 28 29 30 31 56 57 58 59 60 61 62 63
mlx4_3:              24 25 26 27 28 29 30 31 56 57 58 59 60 61 62 63
*****

```

- 4) The source node generates file groups according to the physical storage device they reside on, and then, allocate reader/sender threads according to storage/network device type and the current capacity. At last, the source node will send the file grouping and thread allocation result to the sink node with “*flist*” command in mdtmApp protocol.

```

=====
File grouping result:
TaskGroup 16768 on sdy
File 1: /data/node0/Sfile0
File 2: /data/node0/hundredg0
TaskGroup 17168 on sdax
File 1: /data/node2/Sfile2
File 2: /data/node2/hundredg2
=====

```

```

=====
Thread allocation result:
TaskGroup 16768: Number of readers is 8, Number of senders is 4
=====
Thread allocation result:
TaskGroup 17168: Number of readers is 8, Number of senders is 4
=====

```

- 5) The sink node agrees on the grouping and allocation result, and fork one process to handle each individual task group, this also happens on the source node. The task group process then creates multiple storage I/O and network I/O threads accordingly, and all the I/O threads will be bind to a specific CPU core using the mdm middleware interface *mdtm_schedule_threads(mdtm_thread_desc_t *desc, int thread_num).*

```

Device CPU:Cost
eth4 0:48 1:48 2:48 3:48 4:48 5:48 6:48 7:48 8:69 9:69 10:69 11:69 12:69 13:69 14:69 15:69 16:6 17:6 18:6 19:6 20:6 21:6 22:6 23:6
9:69 30:69 31:69
CPUs Costs Path
16 6 (Core 16)==>(Socket 2)==>(NUMANode 2)==>(Bridge Host->PCI 10)==>(Bridge PCI->PCI 12)==>(PCI 15b3:1003 11)==>(eth4)
17 6 (Core 17)==>(Socket 2)==>(NUMANode 2)==>(Bridge Host->PCI 10)==>(Bridge PCI->PCI 12)==>(PCI 15b3:1003 11)==>(eth4)
18 6 (Core 18)==>(Socket 2)==>(NUMANode 2)==>(Bridge Host->PCI 10)==>(Bridge PCI->PCI 12)==>(PCI 15b3:1003 11)==>(eth4)
19 6 (Core 19)==>(Socket 2)==>(NUMANode 2)==>(Bridge Host->PCI 10)==>(Bridge PCI->PCI 12)==>(PCI 15b3:1003 11)==>(eth4)
20 6 (Core 20)==>(Socket 2)==>(NUMANode 2)==>(Bridge Host->PCI 10)==>(Bridge PCI->PCI 12)==>(PCI 15b3:1003 11)==>(eth4)
21 6 (Core 21)==>(Socket 2)==>(NUMANode 2)==>(Bridge Host->PCI 10)==>(Bridge PCI->PCI 12)==>(PCI 15b3:1003 11)==>(eth4)
22 6 (Core 22)==>(Socket 2)==>(NUMANode 2)==>(Bridge Host->PCI 10)==>(Bridge PCI->PCI 12)==>(PCI 15b3:1003 11)==>(eth4)
23 6 (Core 23)==>(Socket 2)==>(NUMANode 2)==>(Bridge Host->PCI 10)==>(Bridge PCI->PCI 12)==>(PCI 15b3:1003 11)==>(eth4)
CPU Affinity:
PID=51597 TID=4078458624
CPU 16 is set

```

- 6) Multiple task groups will be transferred simultaneously. Each task group process with “*getg*” command in mdmApp protocol to request files in specific task group, and “*get*” command to request a particular file.

```

--> Start to transfer task group 16768
mdtmApp: Creating /dev/null/Sfile0
Device CPU:Cost
--> Start to transfer task group 17168
mdtmApp: Creating /dev/null/Sfile2

```

3.4. Collaborating Environment

3.4.1. Major Activities and Progress

To coordinate the development work from across the two remote labs, BNL and FNAL, we established a set of tools and environment to share information and synchronize our progress.

Autotools: This part was successfully completed. The GNU Autotools are standard software development tool set. They have been established before the coding work started.

RedMine/GIT: This part was successfully completed. The MDTM project used RedMind and GIT as the software version control tools.

Sharepoint and Public website: This part was successfully completed. The MDTM project took use of the Sharepoint and public website to share documents and release project information.

3.4.2. Significant Results

See Appendix C to see the snapshot of the collaborating Environment.

4. Next Steps

The application development team continues their software implementation currently, and has identified several tasks to be completed in the near future. The following ones are the most important tasks to be completed in two months (by the end of May, 2014).

- Multi-rail parallel data transfer (Part of transmission/access module)
 - Mapping multiple groups of files to parallel network interfaces
 - Locality and NUMA affinity are considered
 - Load-balancing between groups and multiple interfaces
- Optimization of file transfer (Part of preprocessing module)
 - Finding the location of files in storage systems
 - Sorting/reordering the files for better storage access performance
 - LVM (logical volume management) of Linux: considering how to handle files spanning across multiple physical disks
- Design of test scenarios, including different storage devices, multiple NICs, different types of workloads (Also part of interface module)

In the next step, MDTM middleware is planned to,

- Integrate the real-time status of the system devices into the scheduling algorithm. The current cost function mainly counts on the topology like distance information. We are working on adding the real-time status like CPU loading, memory usage, IO traffic conditions and etc. to make the scheduling decision more complete and accurate.
- Implement the monitor features that enable the end users to have a clear vision of the underlying activity and manage the middleware and applications in an easy way.

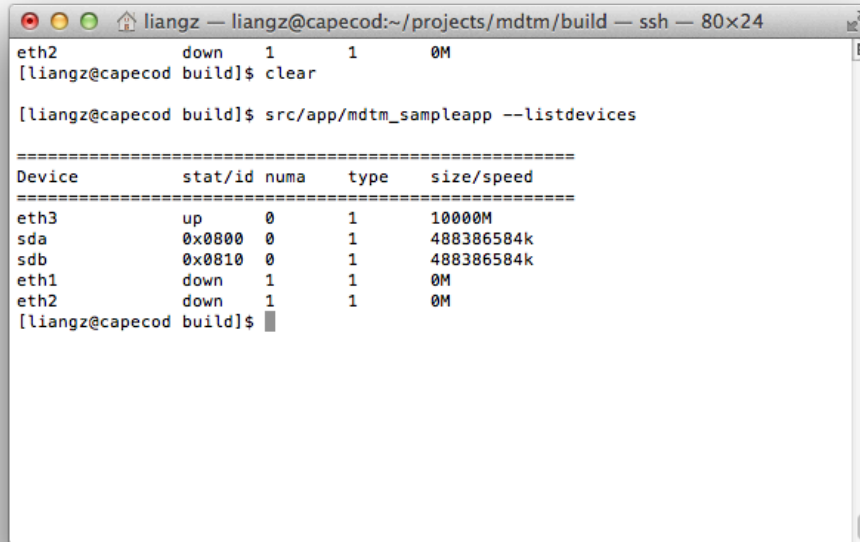
Appendix A: MDTM Application Example Configuration File

The example file below shows the syntax of the configuration file, as well as the device mapping information on one of our testbed NUMA host.

```
# MDTMApp configuration file syntax
# Storage: [device purpose] [major] [minor] [logical name] [physical name] [storage type] <bandwidth capacity>
# Network: [device purpose] [local IP address] [interface name] [network type] <bandwidth capacity>
stor 0      5    zero  zero  mem   100
stor 65     128  sdy   sdy   ssd    10
stor 67     16  sdax  sdax  ssd    10
stor 67     32  sday  ib1    san    40
stor 8       1  sda1  sda   hdd     2
stor 8       2  sda2  sda   hdd     2
stor 8       3  sda3  sda   hdd     2
stor 65     177  sdab1 sdab  hdd     2
stor 65     178  sdab2 sdab  hdd     2
stor 65     179  sdab3 sdab  hdd     2
stor 9       1  md1   md1   sraid1  2
stor 9       2  md2   md2   sraid1  2
stor 9       0  md0   md0   sraid1  2
net  130.245.191.122 eth0  eth  1
net  192.168.13.11  eth4  roce 40
net  192.168.14.11  eth6  roce 40
net  192.168.150.11 ib0   ib   56
```

Appendix B: MDTM Middleware Example Outputs

Devices Status: MDTM system profiling module outputs the devices status.

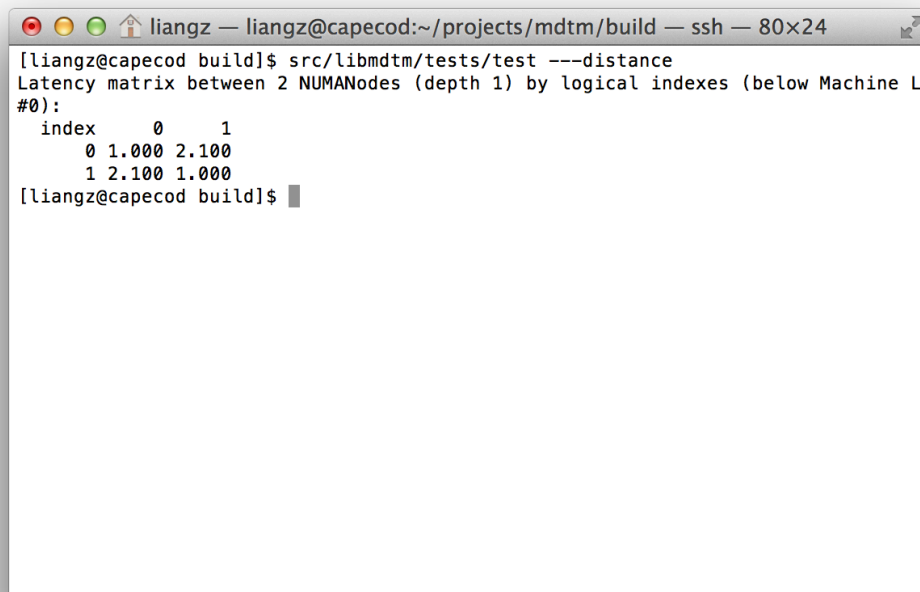
A terminal window titled 'liangz — liangz@capecod:~/projects/mdtm/build — ssh — 80x24'. The user enters 'clear' and then 'src/app/mdtm_sampleapp --listdevices'. The output is a table with columns: Device, stat/id, numa, type, and size/speed. The table lists devices eth3, sda, sdb, eth1, and eth2 with their respective status, IDs, NUMA nodes, types, and sizes/speeds.

```
eth2      down    1      1      0M
[liangz@capecod build]$ clear

[liangz@capecod build]$ src/app/mdtm_sampleapp --listdevices

=====
Device      stat/id  numa   type   size/speed
=====
eth3        up       0       1     10000M
sda         0x0800  0       1     488386584k
sdb         0x0810  0       1     488386584k
eth1        down     1       1       0M
eth2        down     1       1       0M
[liangz@capecod build]$
```

NUMA Node Distances: MDTM profiling module outputs the NUMA node distances.

A terminal window titled 'liangz — liangz@capecod:~/projects/mdtm/build — ssh — 80x24'. The user enters 'src/libmdtm/tests/test ---distance'. The output shows a latency matrix between 2 NUMA nodes (depth 1) by logical indexes (below Machine L #0). The matrix is a 2x2 table with values 1.000 and 2.100.

```
[liangz@capecod build]$ src/libmdtm/tests/test ---distance
Latency matrix between 2 NUMANodes (depth 1) by logical indexes (below Machine L
#0):
  index    0    1
    0 1.000 2.100
    1 2.100 1.000
[liangz@capecod build]$
```

Scheduling and Binding CPU: MDTM scheduler finds the CPU with lowest cost and binds the target thread to that CPU.


```
liangz — liangz@capecod:~/projects/mdtm/build — ssh — 94x26

=====
Scheduling from one device
=====
Device  CPU:Cost
eth1    0:27 1:27 2:27 3:27 4:27 5:27 6:27 7:27 8:6 9:6 10:6 11:6 12:6 13:6 14:6 15:6
CPUs    Costs  Path
8       6      (Core 8)==>(Socket 1)==>(NUMANode 1)==>(Bridge Host->PCI 4)==>(Bridge PCI->PCI
5)==>(PCI 8086:1521 4)==>(eth1)
9       6      (Core 9)==>(Socket 1)==>(NUMANode 1)==>(Bridge Host->PCI 4)==>(Bridge PCI->PCI
5)==>(PCI 8086:1521 4)==>(eth1)
10      6      (Core 10)==>(Socket 1)==>(NUMANode 1)==>(Bridge Host->PCI 4)==>(Bridge PCI->PC
I 5)==>(PCI 8086:1521 4)==>(eth1)
11      6      (Core 11)==>(Socket 1)==>(NUMANode 1)==>(Bridge Host->PCI 4)==>(Bridge PCI->PC
I 5)==>(PCI 8086:1521 4)==>(eth1)
12      6      (Core 12)==>(Socket 1)==>(NUMANode 1)==>(Bridge Host->PCI 4)==>(Bridge PCI->PC
I 5)==>(PCI 8086:1521 4)==>(eth1)
13      6      (Core 13)==>(Socket 1)==>(NUMANode 1)==>(Bridge Host->PCI 4)==>(Bridge PCI->PC
I 5)==>(PCI 8086:1521 4)==>(eth1)
14      6      (Core 14)==>(Socket 1)==>(NUMANode 1)==>(Bridge Host->PCI 4)==>(Bridge PCI->PC
I 5)==>(PCI 8086:1521 4)==>(eth1)
15      6      (Core 15)==>(Socket 1)==>(NUMANode 1)==>(Bridge Host->PCI 4)==>(Bridge PCI->PC
I 5)==>(PCI 8086:1521 4)==>(eth1)
CPU Affinity:
      CPU 9 is set
[liangz@capecod build]$
```

Path Finding between Any Devices: MDTM scheduler has the intelligence to find the internal path from any device to another device in the target system.

```
liangz — liangz@capecod:~/projects/mdtm/build — ssh — 160x48

[liangz@capecod build]$ src/libmdtm/tests/test --path eth1 sda
=====
Print Path Between Devices
=====
(eth1)==>(PCI 8086:1521 4)==>(Bridge PCI->PCI 5)==>(Bridge Host->PCI 4)==>(NUMANode 1)==>(NUMANode 0)==>(Bridge Host->PCI 0)==>(PCI 8086:1d02 3)==>(sda)
[liangz@capecod build]$ src/libmdtm/tests/test --path eth1 eth2
=====
Print Path Between Devices
=====
(eth1)==>(PCI 8086:1521 4)==>(Bridge PCI->PCI 5)==>(PCI 8086:1521 5)==>(eth2)
[liangz@capecod build]$ src/libmdtm/tests/test --path sda sdb
=====
Print Path Between Devices
=====
(sda)==>(PCI 8086:1d02 3)==>(sdb)
[liangz@capecod build]$ src/libmdtm/tests/test --path eth2 sdb
=====
Print Path Between Devices
=====
(eth2)==>(PCI 8086:1521 5)==>(Bridge PCI->PCI 5)==>(Bridge Host->PCI 4)==>(NUMANode 1)==>(NUMANode 0)==>(Bridge Host->PCI 0)==>(PCI 8086:1d02 3)==>(sdb)
[liangz@capecod build]$
```

Appendix C: MDTM Collaborating Environment

SharePoint: Sharing documents and releasing updates of the MDTM project.

Multicore-Aware Data Transfer Middleware ▶ Home

MDTM[About](#)[Download](#)[Docs](#)[Developer](#)[Blog](#)[Search](#)

MDTM: A Multicore-Aware Data Transfer Middleware Project

The MDTM project is dedicated to developing the next generation of high-performance data movement tool.

The MDTM project aims to accelerate data movement at multiore systems. It addresses inefficiencies in existing data movement tools when running on multicore systems by harnessing multicore parallelism to scale data movement on end systems. Essentially, MDTM consists of two components: *data transfer applications/tools* and *middleware*.

The MDTM project will be carried out at Fermi National Acceleration Laboratory (Fermilab) and Brookhaven National Laboratory (BNL). It is sponsored and funded by **DOE Advanced Scientific Computing Research (ASCR) Program**.

>> [see more](#)

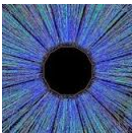
```
graph TD
    A[MDTM Data Transfer Applications/Tools] <-->|Access services| B[MDTM Middleware Services]
    B <-->|Access services| C[OS Services]
    C <-->|Access services| D[Hardware]
```

Announcements

Title	Modified
FNAL and BNL Hold The First On-site Meeting at New York	12/11/2013 4:43 PM

[Add new announcement](#)

MDTM Highlight



The researchers from FNAL and BNL hold a two-day-long joint working meeting during Dec. 9 to 10 at Brookhaven National Lab at New York.

They discussed designs and issues on the undergoing project, coordinated the development procedures and tools in two labs and setup the development milestones for the year of 2014.

Dec. 11, 2014

Fermilab

Copyright

2013

Redmine/GIT: The version control and bug tracking of MDTM source code.

MDTM

Search: MDTM

[Overview](#)[Activity](#)[Issues](#)[New issue](#)[Gantt](#)[Calendar](#)[News](#)[Documents](#)[Wiki](#)[Files](#)[Repository](#)[Code reviews](#)[Settings](#)

root @ master

Statistics | Branch: master | Revision:

Name	Size
doc	
src	
Makefile.am	69 Bytes
README	0 Bytes
config.h.in	557 Bytes
configure.ac	535 Bytes

Latest revisions

#	Date	Author	Comment	Code reviews
0570f6f7	12/09/2013 09:14 am	Liang Zhang	Adding files and dirctory.	No reviews:Assign
7ade0119	12/09/2013 08:11 am	Liang Zhang	First version.	No reviews:Assign
4bc3986f	12/09/2013 07:43 am	Liang Zhang	delete README	No reviews:Assign
b27118d0	11/26/2013 05:02 pm	zlion	First version.	No reviews:Assign

[View differences](#)

[View all revisions](#) | [View revisions](#)

Also available in: Atom

Appendix D: MDTM Library API Functions

Public Member Functions

<code>mdtm_sched_spt</code> (const <code>mdtm_node_t</code> * <code>mdtm_tree</code>)
<code>~mdtm_sched_spt</code> ()
int <code>mdtm_sched_cpu</code> (char * <code>srcdevice</code> , char * <code>dstdevice</code>)

Public Attributes

const <code>mdtm_node_t</code> * <code>mdtmtree</code>
--

Static Public Attributes

static const unsigned <code>cost_max</code> = (unsigned)-1
--

Private Member Functions

int <code>mdtm_dijkstra</code> (const <code>mdtm_node_t</code> * <code>spt</code> , char * <code>device</code> , const <code>cpu_cost_t</code> * <code>cpucosts</code>)
void <code>mdtm_getmincost</code> (const <code>mdtm_node_t</code> * <code>tree</code> , <code>mdtm_node_t</code> ** <code>minnode</code> , unsigned * <code>mincost</code>)
unsigned <code>mdtm_getnodewithmincost</code> (<code>mdtm_node_t</code> ** <code>node</code>)
unsigned <code>mdtm_dijkstra</code> (<code>mdtm_node_t</code> * <code>cpu</code> , const char * <code>device</code>)
void <code>mdtm_updateneighborcosts</code> (<code>mdtm_node_t</code> * <code>node</code>)
int <code>mdtm_calc_costs</code> (const char * <code>device</code> , <code>cpu_cost_t</code> * <code>cpucosts</code>)
void <code>mdtm_resetcostnflag</code> (<code>mdtm_node_t</code> * <code>tree</code>)
void <code>mdtm_print_path</code> (<code>mdtm_node_t</code> * <code>src</code> , <code>mdtm_node_t</code> * <code>dst</code>)
bool <code>mdtm_get_path</code> (<code>mdtm_node_t</code> * <code>src</code> , <code>mdtm_node_t</code> * <code>dst</code> , <code>mdtm_node_t</code> ** <code>path</code> , int * <code>depth</code> , int * <code>maxdepth</code>)
int <code>mdtm_create_spt</code> (void)
void <code>mdtm_connect_numas</code> (void)
int <code>mdtm_destroy_spt</code> ()

Functions

void <code>mdtm_show_version</code> ()
int <code>mdtm_init</code> ()
void <code>mdtm_deinit</code> ()
int <code>mdtm_getcpuaaffinity</code> (char * <code>devicename</code> , <code>mdtm_cpuset_t</code> ** <code>cpuset</code>)
struct <code>mdtm_FileMap</code> * <code>mdtm_findfilemap</code> (const char * <code>pathname</code>) Find the file distribution according the file name. More...
void <code>mdtm_destroyfilemap</code> (struct <code>mdtm_FileMap</code> * <code>map</code>) Release associated resources to <code>mdtm_FileMap</code> object. More...
void <code>mdtm_printpath</code> (char * <code>srcdev</code> , char * <code>dstdev</code> , void * <code>tree</code>)
void <code>mdtm_printschedresult</code> (char * <code>srcdev</code> , char * <code>dstdev</code> , void * <code>tree</code>)
int <code>mdtm_schedule_threads</code> (<code>mdtm_thread_desc_t</code> * <code>thread_descs</code> , int <code>numofdescs</code>) Schedule and pin user threads on CPU cores. More...
void <code>mdtm_sched_per_single_device</code> (<code>mdtm_thread_desc_t</code> * <code>desc</code> , void * <code>tree</code>)
int <code>mdtm_getpcidevices</code> (struct <code>mdtm_device_s</code> ** <code>devices</code>)